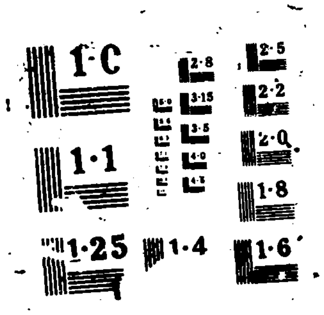
AD-A193 654 SEPARABILITY AND SECURITY MODELS(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) C T SENNETT ET AL. NOV 87 RSRE-87020 DRIC-BR-105251 1/1

UNCLASSIFIED F/G 12/8 NL

END
DATE
FILMED
5 8[illegible]


1·0
2·8
2·5
3·15
2·2
3·5
2·0
1·1
4·0
1·8
1·25
1·4
1·6

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 87020

Title: Separability and security models

Authors: C T Sennett, R Macdonald

Date: November 1987

Summary

This report presents a review of security models and gives a detailed mathematical description of the property of separability, that is the property that certain software can be treated as though it were being obeyed in isolated computers. The description uses the specification language Z, and the proofs required are given in detail.

SEPARABILITY AND SECURITY MODELS

CONTENTS

ACKNOWLEDGEMENT

We would like to acknowledge the assistance of Mr J Pamnani of SCICON Ltd, with some early work concerned with the formalisation in Z of security models.

§1. Introduction

A major element of the programme for increasing the trustworthiness of computers used for processing classified information has been the introduction of mathematical methods, in particular, the use of formal logic to specify security properties and to verify that they obtain within a given software implementation. This leads to the requirement to formalise security properties and use the formalisation to express the security policy which a given system should conform to. This formalisation has proved to be surprisingly difficult, particularly in the matter of relating the formal specification to the intuitive concepts of security, which have proved to be elusive. This report is concerned with one particularly fundamental security concept, namely that of separation. Historically, security has been achieved by means of isolation, that is, the separation of the classified matter from the unclassified, so this concept ought to have an important role to play within secure computer systems. The specification of this important idea is not quite so simple as it might appear and the formal treatment is devoted to giving a definition of separation in a sufficiently precise manner so that the assumptions and implications in what is being defined are made clear.

In this treatment we have followed the work of Rushby [1981] and it is part of the motivation for this report to give this significant work more exposure. Our treatment uses the specification language Z [see Hayes 1987, Sufrin 1983] and another motivation for the report is to show how a formal language such as Z can make the proofs easier to follow and the concepts more precise. As a result of the use of Z, the exposition has more formality than Rushby's, the proofs are slightly more general and have been given in sufficient detail for it to be credible that the proof steps could be checked mechanically. The report commences with the historical background and a discussion of what part separability can play in the general programme for producing secure systems, then relates it to other techniques for defining security models and concludes with a discussion of how separability may be proved in practice, with particular application to the secure communications processor SCP2 [see Harrison and Andrews, 1986, Bottomley 1986].

§2. Security models

A security model is intended to capture the security requirements in a sufficiently general way for it not to pre-judge the design and in a sufficiently simple way for the meaning to be clear. The implemented system must be shown to conform to the model, so the model itself has to be based on some theory of computation, usually a finite state machine: this computational model must then be translated into the model used by the design and implementation languages, so that the security requirements expressed by the model can be checked against the properties of the design or implementation.

The archetypal model is that due to Bell and LaPadula [1976] for the Multics system. The security requirement expressed by this model is a mandatory access control policy. The computer is viewed as a collection of objects which are regarded as information storage containers; information in the containers is only available when they have been accessed (opened for reading or writing) after which it may be transferred between them as a result of the action of subjects, programs and processes which are surrogates for the human users. Consequently subjects can be given clearances, derived from the human users they represent, and objects can be given classifications according to the information they contain. The model uses a state machine description in which the state consists of a record of which subjects are reading or writing which objects, and the transitions of the machine stand for changes in access state. The access control policy is then given by two rules which specify the allowed transitions. These rules express the fact that a subject should have the appropriate clearance to read an object and if it has write access to an object, the object should have at least the classification of any other object which the subject can read.

To use this model, it is necessary to decide what in the real system corresponds to objects (for example, files, devices and so on), what to the subjects (mainly the untrusted user processes) and what to accesses (open file operations etc). From these correspondences, it is possible to work out what are the security rules for each of the operations identified as accesses. This process of instantiation is entirely informal, but this seems to be inevitable for all approaches to security models, which are required to be generic. The informality introduces a dependence on the intuition of the designers which may bring in vulnerabilities; in particular, the set of subjects and objects should be complete, in the sense of representing all active and all information storage entities respectively, otherwise major information flow channels may be left uncontrolled. It would undoubtedly be better if the model could be treated as a top level specification and the process of instantiation treated as an example of refinement, with the security requirements being computed as proof obligations. However, the informal process of instantiation is not difficult to carry out, and the proof requirements for the operations identified are fairly easily related to the model and understood.

The model has been widely used but has also attracted serious criticism, culminating in the paper by McLean [1987], which should be consulted for further details. The basic problem with this model is that the formalism has not brought out the security requirements, and in particular, McLean [1985] has shown that it is possible to prove the central theorem in Bell and LaPadula's paper about a manifestly insecure system. The conclusion of this is that the Bell and LaPadula model captures a notion of security corresponding to contemporary practice in secure systems, rather than some formal definition of security itself. This is at once a strength in terms of being easily understandable and easy to apply, but is also its weakness in terms of having unsuspected loopholes. In particular, there are two deficiencies of the model, one to do with changing the security level of an object, the other to do with covert channels.

The counter example to the model in the paper by McLean is simply one in which objects are downgraded to the lowest security level on every opportunity: this satisfies the model, but is manifestly insecure. Proponents of the model have promptly added a "tranquillity principle" to the model, which states that objects being accessed may not have their security levels changed. This however is not at all satisfactory on philosophical grounds. The addition of a restriction to a definition in this way has been called "exception barring" by Lakatos [1976] and arises from "concept stretching" as one realises that security is concerned not only with information flow between containers, but also with the security labels on the containers themselves. The question arises as to whether these ad hoc extensions of the definition of security are sufficient, or whether others will be found which show the definition to be inadequate.

In the case of the Bell and LaPadula model, the extension is plainly inadequate as objects could be downgraded while not being accessed. Furthermore, taking a step back from the information flow aspects, security is obviously concerned with a whole range of concepts, such as authentication, identification and integrity of audit trails, to name only a few, so one can quite legitimately ask why only the information flow aspects are being formalised. The answer to this question must lie in the perceived threats to the operational system as a result of which one presumably assigns over-riding importance to the information flow aspects; but this is a human judgement which can only be made within the context of a proper risk assessment. Ideally, a security model should be derived from a formalisation of the set of security concepts judged important as a result of a human factors analysis of the operational system. [See Dobson 1987 for some interesting work in this area.]

The other problem with the Bell and LaPadula model is concerned with covert channels: the operations of the model itself, or rather of the functions which implement them, can give rise to information flows which may be exploited to breach security. This problem tends not to be significant in practice, but one again has the philosophical problem that the original definition, even restricted to information flows, is incomplete. Practically, one would prefer to discover the covert channels during the formalisation of the design, and possibly relax the requirement, than search for them in the implemented system and attempt to counter their effects.

As far as information flow requirements are concerned, both problems are addressed by considering the input-output behaviour of a system, rather than the properties of the internal machine states. This is preferrable, since a description in terms of machine states must convey some idea of an implementation, an undesirable characteristic in a security model, which is a top-level specification. In addition, a specification in terms of machine states will nearly always involve covert channels. Separability was the first security concept to be expressed in terms of input-output behaviour and simply states that changes in inputs at one security level should have no effect on outputs at any other level, a pleasing general statement which undoubtedly captures the security concept of separation. Shortly after Rushby published his work Goguen and Meseguer published their work on non-interference [1982, 1984], which, like separability, is expressed in terms of input-output behaviour. Non-interference is a relation between selected inputs and outputs such that variations on the inputs have no effect on the selected outputs. Non-interference is more general than separability because separability may be expressed in terms of non-interference assertions but not vice-versa. However, this report concentrates on separability partly because it is an easier concept to understand and partly because the mathematical description developed by Rushby has some interesting properties.

Regardless of the security model employed, it becomes necessary to show at some point that the implementation conforms to it. This will usually involve the information flow properties of the implemented software and it is tempting to define the security model also in information flow terms: one can simply require that if an operation causes data to flow from A to B the classification of B should dominate that of A, thus avoiding the concept of users altogether and achieving a much more transparent description of the security properties required. An analysis of security along these lines may be found in Denning [1976, 1977]. The concept of information flow is useful, but it is difficult to apply within a trusted computer system. This is because the security requirement is not to prove some property about the total system, but rather about one part of it which must constrain the behaviour of the rest. In effect, trusted and untrusted software is running alternately and the trusted software is required to ensure that no matter what information flows the untrusted software brings about, security is not breached. The trusted software achieves this by running untrusted software for one user at a time; if this fact is ignored there is a plain breach of security because the system will have many files open for reading and writing at different security levels at once while untrusted software is running. In such a system, security is preserved by not allowing one user access to another's open files, so the concept of a user is inevitable in the security model.

Associated with this is the concept of a current security level, which also seems to be inevitable in dealing with an information flow policy. To illustrate this consider the following extract from a program designed to work either in exercise mode or in the real world, where the separation of real and exercise data is brought about using security levels:

```
IF security level implies exercise
THEN open exercise file
ELSE open real file
FI;
calculate data
```

A simple application of information flow makes data dependent on the exercise file, the real file and the security level, containing the information that this is an exercise; consequently the classification of data must dominate both real and exercise, which may not be allowable and is certainly not what is required which is rather that data is classified exercise in exercise mode and real in real mode. Thus information flow must be applied within a context in which users and security levels have been established using a separate security model to specify the policy. This is the basis of the tools developed by Feiertag et al [1977] and more recently the MLS formula generator in enhanced HDM [see Rushby 1985]; information flow must form the basis of any proof of conformity of software with a security policy.

The proof of conformity will itself need to be structured. The information flow aspects of the security policy should be expressed quite generally in terms such as "Secret users should not be allowed to see Top Secret data", and the security model will reflect this policy, in these general terms. The instantiation of the model will be in terms of the operations of the system and the instantiation will incur the proof obligations to ensure that the information flow policy expressed by the model is upheld. In anything but toy systems, the instantiation does not form a satisfactory implementation specification and it is necessary to refine the design towards the actual implementation. Unfortunately the rules of refinement as usually employed do not necessarily maintain the security properties proved at the more abstract level. Ideally, a refinement rule maintaining the security properties should be developed, but this is not usually undertaken and the security properties are maintained informally.

In this process of refinement, separability has an important role to play. At a primitive enough level in the development, the information flow requirements will always reduce to separability requirements: one must establish the isolation of individual objects from each other so that control of access to the object is all that is required to maintain security. Thus for a secure filestore it becomes important to establish that a file at a given security level is isolated from files at every other level, so that a filing system can be considered to be a series of filing systems each operating at one security level. This is precisely the purpose of the separability concept, so the technique of proving separability is highly relevant at this level of the implementation. It is also worth emphasising that this level of proof is neither trivial nor unimportant: in practice, security weaknesses have come about because of weaknesses, such as privileged access mechanisms, occuring at primitive levels in an operating system, rather than weaknesses in the overall design, which is the subject of the high level models.

§3. Proof of separability

Having given some indication of the role that separability can play, a specification of the concept in Z will now be given. The formalisation is based on the use of finite state machines whose input-output behaviour is separable: that is, the input and output can be arranged into streams such that the alteration of one stream of input only affects one stream of output. The different streams of data input and output will be indicated by giving them a label drawn from a given set of labels; the actual data input and output will also be indicated by given sets as follows:

$$[\mathrm{LABEL},\ I,\ O]$$

A member of the set O corresponds to the data output for a given label at any one time and correspondingly for data input. The total output at any one time will therefore consist of a collection of Os, each indexed by a different label. It is useful to give a syntactic definition of a generic set to represent this, as follows:

$$\begin{array}{|l} [X] \\ \mathrm{coll}\ X \triangleq \mathrm{LABEL} \rightarrow X \end{array}$$

This may be used to define the input and output of the state machine:

$$\mathrm{IN} \triangleq \mathrm{coll}\ I; \quad \mathrm{OUT} \triangleq \mathrm{coll}\ O$$

With this definition, given an $in : \mathrm{IN}$, the component of input labelled l is simply $in\ l$.

Transitions of the state machine are brought about by operations. In any given state, observations can be made of the machine, resulting in some member of the set OUT. This can be represented by the schema OP, generic in the machine state.

$$\begin{array}{|l} \mathrm{OP[STATE]} \\ \hline st,\ st' : \mathrm{STATE} \\ out : \mathrm{OUT} \\ \hline \end{array}$$

This schema applies to state machines in general and is equivalent to Rushby's NEXTSTATE and OUTPUT functions taken together. The advantage of using a schema rather than a function to represent the operation is that all the necessary conditions for the application of the operation, as well as side effects, may be added into this basic schema and represented together. Also, the refinement of this specification later on when details of individual operations are given is more easily expressed using schemas.

The realistic treatment of input behaviour in a finite state machine is rather difficult, because of the temporal aspects. Input is actually autonomous, as are the user processes, and it would be more natural to express the properties required in terms of a formal system such as CSP, which is suitable for expressing and reasoning about autonomous and undetermined behaviour. In Goguen and Meseguer's treatment of non-interference the approach adopted was to treat the input as determined, in which case it can simply be taken as another form of operation. Rushby adopted the approach of considering the input as taking place at the commencement of a run of the finite state machine and then generalising the input process informally. With this approach, the input can be regarded as defining the initial state:

$$\begin{array}{|l} \mathrm{INPUT[STATE]} \\ \hline in : \mathrm{IN} \\ st_0 : \mathrm{STATE} \\ \hline \end{array}$$

A run of the machine will correspond to an INPUT followed by a sequence of operations, in each of which the initial state is given by the result of the preceding operation.

```
RUN[STATE]
  INPUT
  computation : seq OP
  ------------------------------------------
  (hd computation).st = $st_0$
  ∀ i : 1..(#computation-1)
  • (computation(i)).st' = (computation(i+1)).st
```

A machine is identified with the set of runs it is capable of making

```
Machine[STATE]
  runs : ℙ RUN
  ----------------------
  {r : runs • r.in} = IN
```

The constraint requires the machine to have a response for any input: there could however be several different responses for a given input so the machine is not (yet) constrained to be deterministic. For each of these runs, we are interested in the output seen by a given user, which is given by the output function, again generic in the machine state:

```
[STATE]
  output_ : (LABEL × RUN)→seq O
  ------------------------------------------
    output = λ l : LABEL; r : RUN • r.computation ⨾ outl
  where
    outl = λ OP • out l
```

With this formalisation, it is now possible to give a definition of secure isolation. To quote Rushby, the simplest and most natural definition is surely that the results seen by each user should depend only on his own contribution to the input. This can be expressed by the schema:

```
[STATE]
  Machine
  ------------------------------------------
  ∀ l : LABEL; run1, run2 : runs
  • run1.in l = run2.in l ⇒ output(l, run1) = output(l, run2)
```

In other words, if two runs have inputs which are the same as far as a given label is concerned, the output having that label should also be the same.

Although this clearly represents a statement of secure isolation, it is in fact impossibly restrictive as a description of the behaviour of an actual machine, for two main reasons. The first is a rather trivial one to do with the fact that our model of the machine includes runs in which one run is a prefix of another; clearly, in this case, there is no breach of isolation provided the output sequences agree up to the length of the shorter, and this is easy to include within the formalism. The other reason is rather more subtle. The sequence of outputs available at each label contains one element for each operation. In reality, machines actually carry out operations on behalf of one user at a time, so the output sequences for a given label will contain many consecutive elements which are the same as the machine carries out operations at other security levels, that is, for different labels. It would be impossibly restrictive to require the machine to present exactly the same sequence of outputs no matter what the other users were doing. This is dealt with by Rushby by squashing the output sequences wherever adjacent elements of the output sequences are the same and then comparing the squashed sequences up to the length of the shorter. In this case, it is not quite so easy to see that the new statement does in fact capture the concept of isolation, but before discussing this in detail the new statement of secure isolation will be given.

To deal with the first problem, a prefix relation ≤, and a relation ≥ which tests if two sequences agree up to the length of the shorter are defined as follows:

```
[T]
_≤_, _≥_ : seq T ↔ seq T
----------------------------------------
∀ a, b : seq T • a≤b ⇔ ∃ c : seq T • a⁀c = b
∀ a, b : seq T • a≥b ⇔ a≤b ∨ b≤a
```

A function condense will be defined which has the effect of squashing sequences of outputs with adjacent elements the same. For use with the proofs later on, this will be defined in terms of a condenser function which provides a mapping from the indices of the condensed sequence to those of the original sequence.

```
condenser : seq O→(N⇸N)
----------------------------------------
condenser =
    λ x : seq O
    • μ map : N⇸N
      | ∀ old, new : N
        • old = map new ⇔
             old = #x ∨ x(old+1) ≠ x(old)
             new = 1 ∧ ∀ m : 1..(old-1) • x(m) = x(m+1)
             ∨
             new > 1 ∧ ∀ m : map(new-1)+1..(old-1)
                          • x(m) = x(m+1)
      • map
```

```
condense ≙ λ os : seq O • (condenser os) ⨾ os
```

With these definitions, the new isolation condition becomes

```
SecureMachine[STATE]
Machine
----------------------------------------
∀ l : LABEL; run1, run2 : runs
• run1.in l = run2.in l ⇒
    condense output(l, run1) ≥ condense output(l, run2)
```

In considering whether this condition does still represent secure isolation, it becomes evident that the condensation has hidden a timing channel. The first statement of security specifies the temporal behaviour of the operations of the machine in a way which precludes their use for signalling and this is relaxed in SecureMachine. This is quite legitimate as no timing constraints have been imposed on the individual operations, so it would in any case be possible to communicate information using the differing durations of the operations, thus using features which are outside the formalism. However, timing channels are no very great threat to security, because of their difficulty of exploitation. What is less obvious is that a storage channel has been hidden, brought about by the ability of one user so to arrange things that an operation carried out for another user has no effect, so that the output remains the same. For a given number of inputs there are a number of ways of interleaving the outputs which gives rise to variations which can be used to convey information. This is obscured by the treatment of input, which appears to be discrete, but when this condition is relaxed the condense function will ignore the variation.

In fact, it is impossible for such a channel to be present in any system satisfying the proof of separability to be given later, so it is tempting to make the separability condition itself be the statement of secure isolation. This has not been done because the separability condition is not intuitively obvious: instead the separability condition will be defined and the condition of SecureMachine shown to be a (weaker) consequence.

This raises the interesting philosophical question of precisely what is meant by secure isolation. This problem has been addressed in a number of ways aimed at formalising information flow through a system: examples are the work by Sutherland [1986] and Foley [1987]. In essence, these are concerned with the question, given some output at a given security level, what information at other levels can be deduced from this output and a knowledge of the capabilities of the system as a whole. From this point of view, separability is actually too restrictive. For example, a user may be given a command to find the number of terminals in use on the system. This is a breakdown of separability because when a secure user logs on, he conveys information to an insecure user; the insecure user however may not be able to deduce anything interesting about this fact because he cannot tell if the new user is a secure user or not. And even if he could, this might not be regarded as a breach of security. These are, however, high level problems whereas separability will normally be applied to subsystems and at a relatively primitive level. For these purposes, the statement of separability developed here adequately captures the security requirements.

Having established that which is required to be proved, it now becomes necessary to prove it. Unfortunately, a direct proof of the statement will rarely be possible. This is because the property required is expressed in terms of the properties of all possible sequences of inputs and outputs, rather than a property of the software used to generate the inputs and outputs. For this latter purpose, the separability property needs to be transformed until it can be more directly related to the properties of computer programs. The approach taken is to demonstrate that a separable machine does indeed behave like a collection of separate machines, one for each label. This can then be used as a specification for the program and the proof of separability can then be interpreted as proof of conformity with this specification.

First of all, it is necessary to show how a set of abstract machines can be combined together to make up the composite, labelled, output of the real machine. The labelled, abstract, machine is very similar to the real machine, except that it receives one input from I and makes available one output from O for every state change, rather than collections of I and O. Schemas corresponding to INPUT, OP and RUN may be defined as follows:

LINPUT[LSTATE]

in : I
st_0 : LSTATE

LOP[LSTATE]

st, st' : LSTATE
out : O

```
__LRUN[LSTATE]______________________________
| LINPUT
| computation : seq LOP
|---------------------------------------------
| (hd computation).st = st_0
| ∀ i : 1..(#computation-1)
| • (computation(i)).st' = (computation(i+1)).st
|_____________________________________________
```

For the abstract machines, we are interested in the set of machines which describe the real machine, and consequently the total set of abstract states which is the union of the states of the individual abstract machines. Denoting this total set by the generic parameter LSTATES, each individual abstract machine operates within a set of states which is a subset of this total. Thus the schema to define an abstract machine is generic in the total set, but contains an identifier state to indicate the particular set of states which this machine operates in.

```
__LMachine[LSTATES]_________________________
| state : ℙ LSTATES
| runs : ℙ LRUN[state]
| run : I→runs
| result : runs→seq O
|--------------------------------------------
|   result = λ r : runs • r.computation ⨾ out
| where
|   out ≙ λ LOP•out
|____________________________________________
```

The separability condition has considerably reduced the scope for non-determinism: the predicate in SecureMachine ensures that runs starting from the same input will have the same output, although the order of occurrence of outputs having differing labels may differ. As far as the output for one label is concerned, the machine appears to be determined. In addition, each separable machine is considered to be ideal in the sense of never having a premature termination, so the set of runs are distinct. Consequently, the abstract machines are much more defined than their real counterparts. This is brought about by the constraint on run in the signature which ensures that there is a run for every input and consequently the sequence of outputs is a total function on the input.

Each labelled machine accounts for the input and output at a given label. This is expressed by the relation ▮, defined as follows:

```
==[STATE, LSTATES]==========================
| _▮_ : (LMachine × LABEL) ↔ Machine
|--------------------------------------------
| ∀ LMachine_1; l : LABEL; Machine
| •(θLMachine_1, l)▮θMachine ⇔
|    ∀ r : runs
|    •   condense output(l, r) ≤ condense result_1 rl
|      where
|        rl ≙ run_1 r.in l
|____________________________________________
```

Note the use of ≤ in this definition, rather than ≥ as used by Rushby: the abstract machine must account for any potential output, at a given label, of the real machine while the real machine is allowed to have runs which terminate prematurely. If the ≥ relation were to be used, any machine could be made secure simply by having a set of LMachine which produced no output. The necessity for the application of condense to the abstract output will be discussed later. A real machine is separable if there exists a collection of abstract machines which are compatible in this way:

```
┌─SepMachine[STATE, LSTATES] ────────
│ Machine
│ lms : coll LMachine
├──────────────────────────────────
│ ∀ l : LABEL • (lms l, l)▯θMachine
└──────────────────────────────────
```

The proposition now is, given the existence of a set of abstract machines which are compatible with a given machine as defined by SepMachine above, this machine is secure by the definition of secure isolation:

```
  [STATE, LSTATES]
  ∃ lms : coll LMachine • SepMachine
⊢
  SecureMachine
```

Proof

Expanding the definition of ▯ in the hypothesis gives the schema:

```
┌─[STATE, LSTATES] ─────────────────────────────
│ Machine
│ lms : coll LMachine
├──────────────────────────────────────────────
│ ∀ l : LABEL
│ •    ∀ r : runs
│      •    condense output(l, r) ≤ condense lm.result rl
│        where
│           rl ≙ run r.in l
│   where
│     lm ≙ lms l
└──────────────────────────────────────────────
```

The quantification over all runs allows the following property of pairs of runs with the same input for a given label to be deduced (note that this fixes the run in the labelled machine and hence rl in the predicate):

```
┌─[STATE, LSTATES] ─────────────────────────────────
│ Machine
│ lms : coll LMachine
├──────────────────────────────────────────────────
│ ∀ l : LABEL
│ •    ∀ run1, run2 : runs | run1.in l = run2.in l
│      •    condense output(l, run1) ≤ condense lm.result rl
│           condense output(l, run2) ≤ condense 'm.result rl
│        where
│           rl ≙ run r.in l
│   where
│     lm ≙ lms l
└──────────────────────────────────────────────────
```

The following lemma follows immediately from the definition of ≤:

```
 r, s, t : seq O | r≤t ∧ s≤t
⊢
 r≤s ∨ s≤r
```

The conclusion is simply the definition of ≥ so the schema may be simplified to:

```
┌─[STATE, LSTATES]──────────────────────────────────────
│ Machine
│ lms : coll LMachine
├───────────────────────────────────────────────────────
│ ∀ l : LABEL
│ •   ∀ run1, run2 : runs | run1.in l = run2.in l
│     • condense output(l, run1) ≥ condense output(l, run2)
│   where
│     lm ≙ lms l
└───────────────────────────────────────────────────────
```

The predicate is now independent of lm, so the local definition may be removed, to give the constraint of SecureMachine.□

Proof of separability involves exhibiting a collection of machines having the desired compatibility property, so the next step is to construct this set. Implicit in the definitions so far has been the idea, which is the invariable practice, that the real machine is time-shared between the different labels and in effect runs each of the abstract machines in turn. Given this concept, it is natural to define two functions in the real machine, namely label, which defines which component of the state of the real machine will change next, and ϕ, an abstraction function defined for each label which gives the labelled abstract state corresponding to a given real state. The proof borrows from proof of refinement of specification to design, based on a key idea of Milner [1971] and expounded in Jones' book on software development [1980]. The abstract separated machines are treated as a specification for the system and the specification of the actual machine is treated as a refinement of this, that is, a concrete example of the abstraction. The abstraction function demonstrates the correspondence between the two which is completed by defining how the operations in the concrete machine map on to the abstract machines. For proof of refinement, every operation in the concrete machine must be shown to have a corresponding effect in the abstract machine when viewed through the abstraction function. For proof of separability, the same condition is applied to the operations of the concrete machine for a given label, which should correspond to the appropriate operation in the corresponding abstract machine; but the additional constraint is imposed that the states of all other abstract machines should remain unchanged.

To develop these ideas formally, a time-shared machine with label and abstraction functions will be defined as follows:

```
┌─TSMachine[STATE, LSTATES]────────────────
│ Machine
│ lms : coll LMachine
│ label : STATE→LABEL
│ ϕ : LABEL→STATE↠LSTATES
├──────────────────────────────────────────
│ ∀ l : LABEL • ϕ l ∈ STATE→(lms l).ṡtate
└──────────────────────────────────────────
```

The constraint ensures that the projection function gives the state corresponding to the machine selected by lms. Further constraints on label and constraints to ensure separability will be added in the schemas which follow. First of all, the concrete operations which correspond to the operations of a given labelled machine are simply those operations in which the label function applied to the initial state returns the corresponding label. The refinement constraint on these operations expresses the fact that the abstraction function, applied to the state after an operation in the concrete machine, should recover the corresponding state after an operation in the appropriately labelled abstract machine. This is given by:

```
┌─TSM1[STATE, LSTATES] ─────────────────────
│ TSMachine
├───────────────────────────────────────────
│ ∀ r : runs; l : LABEL
│ •    #copsl ≤ #absops
│      ∀ i : 1..#copsl
│      • (absops i).st' = φ l (copsl i).st'
│    where
│      copsl ≙ r.computation ↾ {OP | label st = l}
│      rl ≙ (lms l).run (r.in l)
│      absops ≙ rl.computation
└───────────────────────────────────────────
```

In this schema, copsl is the sequence of concrete operations carried out for a given label, rl represents the run produced by the abstract machine labelled l, starting with an input equal to the l component of the input to the corresponding run in the concrete machine, and absops gives the sequence of operations applied by the abstract machine in this case. The separability constraint is that as far as the other abstract machines are concerned, nothing has changed during these operations:

```
┌─TSM2[STATE, LSTATES] ───────────────────────────────────
│ TSMachine
├─────────────────────────────────────────────────────────
│ ∀ r : runs; l : LABEL
│ •    ∀ OP | θOP ∈ copsnotl • φ l st' = φ l st
│    where
│      copsnotl ≙ {OP | θOP ∈ rng r.computation ∧ label st ≠ l}
└─────────────────────────────────────────────────────────
```

Finally, the abstract machines must see the same input and produce the corresponding output:

```
┌─TSM3[STATE, LSTATES] ──────
│ TSMachine
├────────────────────────────
│ ∀ r : runs; l : LABEL
│ •    φ l r.st_0 = rl.st_0
│
│      ∀ lop : lops; cop : cops
│      | lop.st = φ l cop.st
│      • lop.out = cop.out l
│    where
│      rl ≙ (lms l).run (r.in l)
│      cops ≙ rng r.computation
│      lops ≙ rng rl.computation
└────────────────────────────
```

The set of machines constructed in this fashion is compatible, in the sense of the ▯ operator. That is, given

```
    TSM[STATE, LSTATES] ≙ TSM1 ∧ TSM2 ∧ TSM3
```

then

```
      [STATE, LSTATES]
      TSM
    ⊢
      SepMachine
```

Proof

Expanding the definition of ▯ in SepMachine gives the schema

```
┌─[STATE, LSTATES] ─────────────────────────────────
│ TSMachine
├───────────────────────────────────────────────────
│ ∀ l : LABEL; r : runs
│ •   condense output(l, r) ≤ condense (lm.result rl)
│   where
│     lm ≙ lms l
│     rl ≙ (lms l).run (r.in l)
└───────────────────────────────────────────────────
```

The sequence of outputs being compared in the predicate are, on the LHS of the ≤ operator, the outputs for label l as produced by the real machine and on the RHS the outputs produced by the abstract machine. The real sequence of outputs is longer because it produces output whenever it runs one of the other labelled machines, but during this time, if separability is to hold, the output for the given label must remain the same: this is the reason for using the condense function on the real output. Unfortunately, the condense function as defined is too drastic: as well as discarding outputs when the real machine is obeying other labelled machines, it also discards repeated output produced by the abstract machine itself and for this reason the condense function has to be applied to the output of the abstract machine as well in order to end up with a comparable sequence.

The proof uses the fact that the output from the abstract machine is a partial condensation of the output of the real machine, in the sense that every output in the abstract machine appears, in the correct order but including repeated outputs, in the real output, and may be repeated an arbitrary number of times. The property of partial condensation is in fact the strong condition for secure isolation which precludes the covert channel discussed earlier. From this the weaker condition given above will be deduced by showing that if one sequence is a partial condensation of another, they will both condense to the same sequence.

The partial condensation occurs as a result of the real machine obeying other labelled machines, so it is possible to calculate the mapping function corresponding to this partial condensation. Before doing this, it is useful to define a schema isolating one run of the machine and a given label:

```
Labelledrun[STATE, LSTATES]
 TSMachine
 l : LABEL; r : runs
 lm : LMachine; rl : LRUN[state];
 cops : seq OP;  absops : seq LOP[state]
 lops : ℙ OP
---------------------------------------------
 lm = lms l
 rl = lm.run (r.in l)
 cops = r.computation ∧ absops = rl.computation
 lops = {OP | label st = l}
```

The mapping function corresponding to the partial condensation maps the outputs in absops to the outputs in cops, filling in the blanks when the labelled machine is not running with the last output. If *f* is the function, then the ouputs in cops are given by the outputs in f;absops. It is easy to see that *f* is determined by whether the concrete state indicates that it is carrying out a labelled operation. The output is given by the output in the initial labelled state until the first labelled operation takes place. After that the output is given by the output after the last labelled operation.

```
mapdef[STATE, LSTATES]
 Labelledrun
 f : ℕ ⇸ ℕ
---------------------------------------------
 dom f = 1..#cops ∧ rng f ⊆ 1..#absops
 ∀ i, j : ℕ
 • j = f i ⇔
     j = 1 ∧ ∃ k : dom cops
               | cops k ∈ lops ∧ ∀ m : 1..(k-1) • cops m ∉ lops
               • i ≤ k
    ∨
     j ≠ 1 ∧ ∃ k : dom cops
               | cops k ∈ lops ∧ k < i
                 ∧ ∀ m : k+1..(i-1) • cops k ∉ lops
               • cops i ∉ lops ∧ j = (f k)+1
                 ∨
                 cops i ∈ lops ∧ j = f k
```

Note that this definition of the mapping function corrects a slip in the definition of the equivalent function used by Rushby which does not treat the initial state correctly. The range of *f* is a subset of the number of abstract operations as a given concrete run may have terminated prematurely. The proposition to be proved, indicating the partial condensation, is

```
Partiallycondensed[STATE, LSTATES]
 mapdef
---------------------------------------------
 ∀ i : 1..#cops • (cops i).out l = (absops f(i)).out
```

This proposition is the strong statement of secure isolation referred to earlier. As can be seen, although it represents a stronger condition than the constraint of SepMachine, its intuitive meaning is not immediately apparent, which is why it is necessary to work in terms of the full condensation. The proposition can be proved as a consequence of the fact that the mapping function can be used to deduce which states in the abstract machine correspond to states in the concrete machine and that the output is a function of the state alone.

```
┌─Correspondingstates[STATE, LSTATES] ──────────────
│ mapdef
├───────────────────────────────────────────────────
│ ∀ i : 1..#cops • φ 1 (cops i).st = (absops f(i)).st
└───────────────────────────────────────────────────
```

The proposition follows immediately from this and TSM3:

```
   [STATE, LSTATES]
   TSM3; Correspondingstates
 ⊢
   Partiallycondensed
```

The correspondence between the states may be derived from TSM1 and TSM2. The proof depends upon the fact that the mapping function for the filtering operator in TSM1 is an inverse for the mapping function f which has just been defined. The mapping function for the filtering operator is given by g in the schema below:

```
┌─compactor[STATE, LSTATES] ─────────────────────────
│ labelledrun
│ g : N ⤔ N
├───────────────────────────────────────────────────
│ ∀ i, j : N
│ • j = g i ⇔
│       cops j ∈ lops
│       i = 1 ∧ ∀ m : 1..(j-1) • cops m ∉ lops
│       ∨
│       i ≠ 1 ∧ ∀ m : g(i-1)+1..(j-1) • cops m ∉ lops
│ cops↾lops = g ⨾ cops
└───────────────────────────────────────────────────
```

There are various possible definitions of the filtering operator ↾; we have taken the view that the first predicate above is an adequate, if somewhat pedestrian, definition. With this definition it is possible to show that f⨾g is an identity relation on the set {i : dom cops | cops i ∈ lops}. The proof of this is rather tedious and has been deferred to the appendix, where it is referred to as theorem 1.

Using this definition of g, it is possible to rewrite TSM1 as

```
─[STATE, LSTATES]──────────────────────
│ TSMachine
│ compactor
├──────────────────────────────────────
│ ∀ r : runs; 1 : LABEL
│ •   #copsl ≤ #absops
│     ∀ i : 1..#copsl
│     • (absops i).st' = Φ 1 (copsl i).st'
│   where
│     copsl ≙ g ⨾ r.computation
│     rl ≙ (lms 1).run (r.in 1)
│     absops ≙ rl.computation
└──────────────────────────────────────
```

For the proof, it is necessary to break the sequence of concrete operations into two, consisting of those operations before and after the first operation for label 1. This occurs at position first in the sequence of concrete operations, as given below:

```
─firstdef[STATE, LSTATES]─────────────────────────
│ mapdef; first : dom cops
├─────────────────────────────────────────────────
│ cops first ∈ lops ∧ ∀ i : 1..(first-1) • cops i ∉ lops
└─────────────────────────────────────────────────
```

Using this, and the revised form of TSM1

```
─[STATE, LSTATES]──────────────────────
│ firstdef
├──────────────────────────────────────
│ ∀ i : first..#cops
│ | cops i ∈ lops
│ • Φ 1 (cops i).st' = (absops f(i)).st'
└──────────────────────────────────────
```

follows immediately from the fact that

```
f ⨾ copsl = f ⨾ g ⨾ cops
          = Id{i : dom cops | cops i ∈ lops} ⨾ cops
          = cops ▷ lops
```

Using the fact that the state after an operation is the same as the state before the next one, this may be transformed into:

```
─[STATE, LSTATES]─────────────────────────────
│ firstdef
├─────────────────────────────────────────────
│ ∀ i : first..(#cops-1)
│ | cops i ∈ lops
│ • Φ 1 (cops(i+1)).st = (absops(f(i)+1)).st
│ ∀ i : first+1..#cops
│ | cops(i-1) ∈ lops
│ • Φ 1 (cops i).st = (absops(f(i-1)+1)).st
│                   = (absops f(i)).st
└─────────────────────────────────────────────
```

The second predicate is a change of the bound variable from i to i+1, while the second equality in this predicate follows from the properties of f.

After the first operation, the 1-projection of the state remains the same whenever label st ≠ 1, and is equal to the state after the last 1-operation.

```
__[STATE, LSTATES]______________________________
| firstdef
|________________________________________________
|
| ∀ i : first+1..#cops
| | cops i ∉ lops
| •   Φ 1 (cops i).st = Φ 1 (cops i).st'
|                     = Φ 1 (cops k).st'
|                     = (absops(f k)).st'
|                     = (absops(f k+1)).st
|                     = (absops(f i)).st
|   where
|     | k : dom cops
|     |__________________________________
|     |
|     | cops k ∈ lops ∧ k < i
|     |   ∧ ∀ m : k+1..(i-1) • cops k ∉ lops
|________________________________________________
```

The first and second equalities follow from repeated use of TSM2, the third from what has just been deduced from TSM1, the fourth by using again the fact that the state after an operation is the same as the state before the next one and the final one from the definition of f.

The initial sequence is dealt with as follows:

```
__firstcase[STATE, LSTATES]______________________
| firstdef
|________________________________________________
|
| ∀ i : 1..first • Φ 1 (cops i).st = Φ 1 (cops 1).st
| Φ 1 (cops 1).st = Φ 1 r.st_0
|    = r1.st_0
|    = (absops 1).st
|    = (absops f(1)).st
| ∀ i : 1..(first-1) • f i = 1
|________________________________________________
```

The first predicate follows from the fact that the 1-component of the state does not change from its initial value until the first 1-operation, as specified in TSM2. The successive lines of the second predicate follow from the definition of RUN, the first predicate of TSM3, the definition of LRUN and the definition of f respectively. The third predicate follows from the definition of f and first.

Combining the conclusions, it follows that

```
[STATE, LSTATES]
TSM
⊢
Partiallycondensed
```

It is now necessary to prove the lemma that if one sequence is a partially condensed version of another then both condense to the same result. This can be stated within the context of the schema

```
┌─Parcon ──────────────────────────────────────────────
│ x, y : seq O
│ f : dom x ⤀ dom y
├──────────────────────────────────────────────────────
│ f 1 = 1
│ ∀ j : dom f | j < #x • f(j+1) = f(j) ∨ f(j+1) = f(j)+1
│ ∀ j : dom x • x(j) = y(f(j))
└──────────────────────────────────────────────────────
```

from which the following lemma is proved as theorem 2 in the appendix:

```
  Parcon
⊢
  condense x = condense y
```

The proof of the lemma depends on the function f being a surjection, so that its range may be identified with the domain of y. For the main theorem the range of f is a subset of the domain of absops, but the lemma may still be applied to that prefix of absops whose domain is equal to the range of f. Consequently the ≤ relation holds between the condensed sequences, from which the main theorem may be deduced.□

§4. Proving separability in practice

In attempting to prove separability, it is necesary to demonstrate that the real machine satisfies the specification TSM. For this it is necessary to define the real and abstract states, the projection function Φ and the label function. Typically, in a machine such as SCP2 [see Harrison and Andrews, 1986, Bottomley 1986], the separation is provided by a separation kernel which runs separate virtual machines using the addressing hardware to ensure isolation. Input and output are controlled by the separation kernel which simulates virtual input and output to the individual virtual machines as a result of operations it carries out on their behalf. To prove separability, what is required is a specification of the constraints which must be imposed on these operations for the secure isolation to be preserved. With this in mind, the real state can be instantiated as a set of virtual machines. As previously, input sets the initial state: a more realistic form of input will be developed later.

Each virtual machine has a data space given the suggestive name ENV and a program consisting of instructions each of which modifies the data space. Functions prog and output are presumed: prog delivers the next instruction to be obeyed and output computes the output available for a given data space:

```
[ENV]
output : ENV → O
INSTRUCTION ≙ ENV → ENV
```

```
┌─ VM ──────────────────────────────────
│ env : ENV
│ prog : ENV ⇸ INSTRUCTION
├───────────────────────────────────────
│ ∀ env : dom prog
│ •   ∃ n : ℕ • Rⁿ env ∉ dom prog
│   where
│     R ≙ λ env : ENV • (prog env) env
└───────────────────────────────────────
```

Termination of the program is modelled by an instruction taking the environment outside the range of application of the program. The constraint ensures that each program gives rise to a finite sequence of obeyed instructions, but is not otherwise relevant to the proof. The real state is given by a collection of VMs and the label for the current VM:

```
┌─ VMSTATE ──────
│ vms : coll VM
│ label : LABEL
└────────────────
```

while the abstraction is that each virtual machine runs in isolation, with operations given by:

```
┌─ VMOP ────────────────────────────
│ ΔVM
│ out : O
├───────────────────────────────────
│   env' = inst env
│ where
│   inst ≙ prog env
│ prog' = prog ∧ out = output env
└───────────────────────────────────
```

The specification for each of the operations of the separability kernel expresses the fact that one VM is run at a time with the others remaining unchanged. The VM which changes must satisfy VMOP:

```
__KOP _______________________________________
| ΔVMSTATE
| out : OUT
|--------------------------------------------
| ∀ l : LABEL | l ≠ label • vms' l = vms l
| ∃ VMOP • θVM = vms label ∧ θVM' = vms' label
| out = λ l : LABEL • output (vms l).env
|____________________________________________
```

Each abstract VM carries out all the operations in its program so it is possible to derive the sequence of operations carried out by an abstract machine from its initial state. This is given by the function opsl defined below:

```
|============================================
| opsl : VM → seq VMOP
|--------------------------------------------
| ∀ vm : VM; computation : seq VMOP
| • computation = opsl vm ⇔
|     vm.env ∉ dom vm.prog ⇒ computation = ⟨⟩
|     vm.env ∈ dom vm.prog ⇒
|       computation = θVMOP cons opsl θVM'
|     where
|       VMOP | θVM = vm
|____________________________________________
```

The abstract operation carried out whenever the real machine carries out a concrete operation is given by an abstraction function:

```
absop ≙ λ KOP • μ VMOP | θVM = vms label • θVMOP
```

The constraints on VMOP ensure that absop delivers a well defined result. Finally, the machine which is a set of virtual machines is given by

```
__VMachine __________________________________
| Machine[VMSTATE]
|--------------------------------------------
| ∀ r : runs; l : LABEL
| • r.computation ↾ {KOP | label = l} ⨾ absop
|       ≤ opsl r.st_0.vms l
| ∀ l : LABEL; run1, run2 : runs
| | run1.in l = run2.in l
| • run1.st_0.vms l = run2.st_0.vms l
|____________________________________________
```

The first constraint ensures that in any real run each virtual machine starts from the beginning and obeys some part of its program. The form of KOP and VMOP ensures that the constraints on machine runs (that the final state of each operation is the initial state of the next) can also be met. The second constraint ensures that the same input seen by any virtual machine will result in it ending up in the same state, as well as ensuring that the machine is always loaded with the same program. From this machine the following TSMachine may be constructed:

```
VMRUN ≙ LRUN[VM]

┌─TSVMachine ──────────────────────────────────────
│ VMachine
│ lms : coll LMachine[VM]
│ label : VMSTATE → LABEL
│ ϕ : LABEL → VMSTATE ⇸ VM
├──────────────────────────────────────────────────
│ label = λ VMSTATE • label
│ ϕ = λ l : LABEL • λ VMSTATE • vms l
│ lms = λ l : LABEL
│         • μ LMachine_1[VM]
│            | state_1 = (ϕ l)⦇VMSTATE⦈
│              run_1 = λ i : I
│                        • μ VMRUN
│                          | in = i
│                            ∃ r : runs | r.in = i
│                            • st_0 = r.st_0.vms l
│                            computation = opsl st_0
│                          • θVMRUN
│              runs_1 = run⦇{r : runs • r.in l}⦈
│            • θLMachine_1[VM]
└──────────────────────────────────────────────────
```

The result of run is uniquely defined because it is determined by the initial state, which is the same when the input is the same.

This machine satisfies the constraints of TSM and hence is separable:

```
  TSVMachine
⊢
  TSM[VMSTATE, VM]
```

Proof

From the definition of VMachine, the following propositions may be deduced:

```
┌──────────────────────────────────────────────
│ Labelledrun[VMSTATE, VM]
├──────────────────────────────────────────────
│   copsl ⨾ absop ≤ opsl r.st_0.vms l
│   copsl ⨾ absop ≤ absops
│   ∀ i : 1..#copsl
│   • (copsl i).st.vms l = (absops i).st
│   ∀ i : 1..#copsl
│   • ϕ l (copsl i).st = (absops i).st
│   ∀ i : 1..#copsl
│   • ϕ l (copsl i).st' = (absops i).st'
│ where
│   copsl = cops ↾ lops
└──────────────────────────────────────────────
```

The first predicate is the first constraint in VMachine, the second follows from the definition of run in TSVMachine, the third from the application of absop, the fourth from the definition of ϕ, and the final one from the fact that st' is determined from st for VMOPs. From this, TSM1 follows immediately. TSM2 follows immediately from the definition of KOP while TSM3 follows from the definition of run and VMachine.□

The properties of the TSVMachine follow from the constraints in KOP and VMOP which in turn determine the properties which should apply to the operations actually obeyed. It is trivial to show that if similar constraints apply to each instruction which the separability kernel implements, the properties hold no matter what programs are loaded into the individual virtual machine, so it is possible to write down a separability specification for each instruction. This is as follows:

```
SepInstruction
ΔVMSTATE
out : OUT
inst : INSTRUCTION
--------------------------------------------
∀ l : LABEL | l ≠ label • vms' l = vms l
  env' = inst env
where
  env ≙ (vms label).env
  env' ≙ (vms' label).env
out = λ l : LABEL • output (vms l).env
```

The artifical form of input must also now be lifted. The problem with real input, as far as this formal description is concerned, is that it takes place autonomously and consequently requires a logic such as CSP to handle. The approach so far has been to treat the sequential temporal behaviour using the properties of sequences and to treat the indeterminacy by using underspecification: for example label' in KOP is not specified so the separability kernel is free to run whichever machine it chooses according to a scheduling algorithm and external events. It is not possible to do this with input, so it is necessary to generalise the model intuitively. This will be done by analogy with output: input is allowed at any time, but the effect it has on each virtual machine is required to be determined solely by the input destined for that machine. In other words a function input will be presumed:

```
input : (ENV × I)→ENV
```

with the following specification for each input:

```
SepInput
ΔVMSTATE
in : IN
--------------------------------------------
∀ l : LABEL
• vms' l = μ VM; VM'
          | θVM = vms l
             ∧ prog' = prog ∧ env' = input(env, in l)
          • θVM'
```

These two schemas complete our specification of separability which may be summarised informally as follows.

> Each instruction carried out for a labelled machine must have no effect on any other machine.
>
> The effect of each instruction carried out for a labelled machine must be determined entirely by the data available to that machine.
>
> During any instruction, the output available should only alter for the labelled machine currently being run, and this changed output should be completely determined by the data available to the current machine.
>
> Changes of machine state brought about by any input at a given label should only affect the corresponding machine, and that effect should be totally determined by the state of the machine and the input.

§5. Conclusions

Presented in this form, the summary of separability can lead to a feeling of exasperation: having been taken through several pages of formal analysis one ends up with conclusions which might have been written down at the beginning because they are, in fact, perfectly obvious. To some extent this feeling is justified because formal methods cannot draw out of the initial statement more than is contained in it already: the main benefit of the analysis is not so much in the conclusion as in the process of arriving at it, for the conduct of the proof should add to the understanding of the problem. For example, the conduct of the proof led to the discovery of covert channels simply as a result of the fact that it was impossible to carry out the proof without explicitly discarding them, which leads to an understanding as to how they arise.

The formal statements are also much more precise than might have been written down simply as a result of an intuitive understanding of the problem. It contains details of precisely when the label variable can be changed for example and the requirements on functional dependence contained in the declarations of INSTRUCTION, out and in are extremely strict and could well have been overlooked, or written in a slightly less restrictive form. The analysis also allows us to deduce that the conditions presented are necessary, that is the minimum consistent with the structure of the software, which itself is more or less inevitable.

These reasons lead to the view that an analysis such as is presented here is an essential underpinning in the development of a theory of security, of which separability must play a part. In spite of this, one is not quite content with the analysis. First of all is the fact, already noted, that Z is not the ideal language for developing the analysis on account of its inability to specify temporal behaviour. While this does not have a major impact on the specification of a primitive property such as separability it does have an impact for higher level properties for which a formalism such as CSP [Hoare 1985] will be necessary. The other criticism one can make of the analysis is that the state machine description, while considerably more realistic than other descriptions in the literature, is still some distance from the behaviour of real machines. A particular failing is in the treatment of backing store where, for example, a disc header is not actually a form of output which can be given any particular label. A correct analysis of this situation requires a state machine description containing elements of persistent storage. This leads on to a further area where research is necessary: the whole point of the separability approach is to prove properties of computers at a primitive level, so it is very natural to want to pursue this analysis down to the level of implementations. Unfortunately, primitive kernel level operations are frequently, and often necessarily, encoded in primitive assembly languages which are not currently susceptible to formal verification.

Finally, it is worth concluding with a few comments on the use of Z in an analysis of this nature. The presentation given here is actually more general than that given by Rushby, whose work is based on the use of total functions, rather than the finite sequences which have been used in this report. It is considerably more precise as it has eliminated the ellipsis dots (...) which occur in Rushby and which proved to contain considerably more information than had at first been apparent. More steps have been given in the proof, but nevertheless the presentation as a whole is of a comparable length, a reflection on the compact nature of the Z notation. It is not appropriate to comment on the relative readabilities of the two presentations except to note that Z does have one advantage over the more informal presentation in Rushby. If one aspect of the development is written in English, but not understood a a considerable difficulty is presented to the reader. However the Z presentation is continuous and parallel to the English discourse. This means that when the discourse is not understood the notation, being complete, is there to help and this can resolve many problems of understanding. It is undoubtedly the case that mathematical presentations of this nature are substantially assisted by the use of a precise machine checkable notation.

REFERENCES

Bell, D E and LaPadula, L J. "Secure computer system: unified exposition and Multics interpretation", Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, Massachussetts, USA, March 1976.

Bottomley, P C. "Technical overview of SCP2 - a multi-level secure communications processor", IEE conference on secure communication systems, London, 1986.

Denning, D E. "A lattice model of secure information flow", Comm ACM vol 19, no 5, pp 236 - 243, May 1976.

Denning D E, and Denning P J. "Certification of programs for secure information flow", Comm ACM vol 20, no 7, pp 504 - 512, July 1977.

Dobson, J E. "The human processes in secure systems analysis", internal report, Computing Laboratory, University of Newcastle upon Tyne, January 1987.

Feiertag, R J, Levitt, K N, and Robinson L. "Proving multi-level security of a system design", Proc 6th ACM Symp on Operating System Principles, ACM SIGOPS Operating System Review vol 11, no 5, pp 57 - 65, Nov 1977.

Foley, S N. "A universal theory of information flow", Proc 1987 Symposium on Security and Privacy, Oakland, California, USA, pp 116 - 122, IEEE Computer Society, 1987.

Goguen, J A and Meseguer, J. "Security policies and security models", Proc 1982 Berkeley Conference on Computer Security, IEEE Computer Society Press, 1982.

Goguen, J A, and Meseguer, J. "Unwinding and inference control", Proc 1984 Symposium on Security and Privacy, Oakland, California, USA. IEEE Computer Society, 1984.

Harrison, P J, and Andrews, M P. "SCP2 - a multi-level secure communications processor", presented at "ONLINE System Security", Wembley, October 1986.

Hayes I. (ed) "Specification Case Studies", Prentice Hall International series in Computer Science, 1987.

Hoare, C A R. "Communicating sequential processes", Prentice Hall International series in Computer Science, 1985.

Jones, C B. "Software development, a rigorous approach", Prentice Hall International, London 1980.

Lakatos, I. "Proofs and refutations - the logic of mathematical discovery", Cambridge University Press, 1976.

McLean J, "A comment on the 'Basic security theorem' of Bell and LaPadula", Information Processing Letters, 20, 1985, pp 67-70.

McLean J. "Reasoning about security models", Proc 1987 Symposium on Security and Privacy, Oakland, California, USA, pp 123 - 131. IEEE Computer Society, 1987.

Milner, R. "An algebraic definition of simulation between programs", Second International Joint Conference on Artificial Intelligence, London 1971.

Rushby, J M. "The design and verification of secure systems", Proc 8th ACM Symposium on Operating System Princilples, Asilomar, California, USA, December 1981. (Available as ACM Operating Systems Review, vol 15, no 5.)

Rushby, J M. "Mathematical foundations of the MLS tool for Revised Special", internal report, Computing Science Laboratory, SRI International, 1985.

Sufrin, B. "Formal system specification - notation and examples", in "Tools and Notations for Program Construction" (Neel ed.), Cambridge University Press, 1983.

Sutherland, D. "A model of information", Proc 9th National Computer Security Conference, NBS and NCSC, Gaithersburg, Marvland, USA, pp 175 - 183, 1986.

APPENDIX - DEFERRED PROOFS

Theorem 1

We have to show that f ; g forms an identity relation on the set of indices for the labelled operations. Start by noticing that the set required is given by the range of g and then expand the composition operator:

```
 _[STATE, LSTATES] ____________________________
|
| compactor
|
|______________________________________________
|
| f;g = { i, m : rng g
|         | ∃ j, l : ℕ | j = l • j = f i ∧ m = g l
|         • i ↦ m
|         }
|     = { i, m : ℕ
|         | ∃ l : dom g | l = f i • m = g l
|         • i ↦ m
|         }
|______________________________________________
```

All the integers l which satisfy the constraint in the explicit construction give equal i and m:

```
[STATE, LSTATES]
compactor
⊢
∀ i, m : rng g; l : dom g | l = f i ∧ m = g l • i = m
```

This may be proved by induction on l. For the base case, one may expand the definition of g for l = 1 to give

```
 _[STATE, LSTATES] ____________________________
|
| compactor
|
|______________________________________________
|
| ∀ i, m : rng g; l : dom g
| | l = f i ∧ m = g l ∧ l = 1
| • cops m ∈ lops
|   l = 1 ∧ ∀ k : 1..(m-1) • cops k ∉ lops
|   ∃ k : dom cops
|   | cops k ∈ lops ∧ ∀ n : 1..(k-1) • cops n ∉ lops
|   • l = 1 ∧ i ≤ k
|______________________________________________
```

and it is possible to satisfy the existential quantification in the predicate with k = m and the only i ∈ rng g which is less than or equal to m is m. For the inductive step the hypothesis is

```
 _[STATE, LSTATES] ____________________
|
| compactor; n : dom g
|______________________________________
|
| ∀ i, m : rng g | n = f i ∧ m = g n • i.= m
|______________________________________
```

Expanding g(n+1) and f i in the next case, and using l ≠ 1 gives

```
┌─[STATE, LSTATES]──────────────────────────────
│ compactor;  n : dom g
├───────────────────────────────────────────────
│ ∀ i, m : rng g
│ | n+1 = f i ∧ m = g(n+1)
│ • ∀ k : (g n)+1..(m-1) • cops k ∉ lops
│   ∃ k : dom cops
│   | cops k ∈ lops ∧ k < i
│     ∧ ∀ m : k+1..(i-1) • cops k ∉ lops
│   • cops i ∉ lops ∧ n+1 = (f k)+1
└───────────────────────────────────────────────
```

The existential quantifier is satisfied with k = g n, from which it immediately follows that the next case may be inferred from the hypothesis. From this the required theorem may be deduced.

Theorem 2

Within the context of this definition,

```
┌─Parcon────────────────────────────────────────────────
│ x, y : seq O
│ f : dom x ↠ dom y
├───────────────────────────────────────────────────────
│ f 1 = 1
│ ∀ j : dom f | j < #x • f(j+1) = f(j) ∨ f(j+1) = f(j)+1
│ ∀ j : dom x • x(j) = y(f(j))
└───────────────────────────────────────────────────────
```

it is necessary to show that:

```
 Parcon
⊢
 condense x = condense y
```

It is useful to work in terms of the condenser functions, defined as below:

```
┌─Condef─────────────────────────────
│ Parcon
│ gx, gy : N ⇸ N
├────────────────────────────────────
│ gx = condenser x ∧ gy = condenser y
└────────────────────────────────────
```

It is required to prove that gx ; x = gy ; y. As x = f ; y, it will be shown that gx ; f is equal to gy.

```
Condef
─────────────────────────────────────────────
gx ⨾ f
    = {i : dom gx; j : rng f
       | ∃ k, l : ℕ | k = gx i ∧ j = f l • k = l
       • i ↦ j
       }
    = {i : dom gx; j : rng f
       | ∃ k : rng gx • k = gx i ∧ j = f k
       • i ↦ j
       }
    = {i : dom gx; j : rng f
       | ∃ k : rng gx
         | j = f k
         • x(k) ≠ x(k+1)
           i = 1 ∧ ∀ m : 1..(k-1) • x m = x (m+1)
           ∨
           i > 1 ∧ ∀ m : gx(i-1)+1..(k-1) • x m = x (m+1)
       • i ↦ j
       }
    = {i : dom gx; j : rng f
       | ∃ k : rng gx
         | j = f k
         • y f(k) ≠ y f(k+1)
           i = 1 ∧ ∀ m : 1..(k-1) • y f(m) = y f(m+1)
           ∨
           i > 1 ∧ ∀ m : gx(i-1)+1..(k-1) • y f(m) = y f(m+1)
       • i ↦ j
       }
    = {i : dom gx; j : rng f
       | ∃ k : rng gx
         | j = f k ∧ f(k+1) = f k + 1
         • y(j) ≠ y(j+1)
           i = 1 ∧ ∀ m : 1..(k-1) • y f(m) = y f(m)+1
                                  ∧ f(m+1) = f(m)+1
           ∨
           i > 1 ∧ ∀ m : gx(i-1)+1..(k-1) • y f(m) = y f(m)+1
                                          ∧ f(m+1) = f(m)+1
       • i ↦ j
       }
    = {i : dom gx; j : rng f
       | ∃ k : rng gx
         | j = f k ∧ f(k+1) = f k + 1
         • y(j) ≠ y(j+1)
           i = 1 ∧ ∀ m : 1..(j-1) • y(m) = y(m+1)
           ∨
           i > 1 ∧ ∀ m : (f gx(i-1))+1..(j-1) • y(m) = y(m+1)
       • i ↦ j
       }
    = {i : dom gx; j : rng f
       | • y(j) ≠ y(j+1)
           i = 1 ∧ ∀ m : 1..(j-1) • y(m) = y(m+1)
           ∨
           i > 1 ∧ ∀ m : (f gx(i-1))+1..(j-1) • y(m) = y(m+1)
       • i ↦ j
       }
```

The equalities in this schema are given by the following rules:

1. Expansion of the functional composition operator

2. Substitution of k for 1.

3. Expansion of $k = gx\ i$

4. Substitute for x using $x = f \mathbin{;} y$

5. Eliminate identically true terms in the universal quantifications occurring when $f(m) = f(m+1)$

6. Change quantification basis using

   $$f(\!|\{m : 1..k-1 \mid f(m+1) = f(m)+1\}|\!) = \{1..f(k)-1\}$$

7. Remove existential quantification as the predicate is independent of k and the constraint may be satisfied

The required property now follows as a result of a simple induction on i.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ....UNCLASSIFIED...

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference REPORT 87020	3. Agency Reference	4. Report Security U/C Classification

5. Originator's Code (if known) 778400	6. Originator (Corporate Author) Name and Location RSRE, ST ANDREWS ROAD, MALVERN, WORCS WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

SEPARABILITY AND SECURITY MODELS

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials SENNETT, C.T.	9(a) Author 2 MACDONALD, R.	9(b) Authors 3,4...	10. Date 1987.12	pp. ref. 30

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This report presents a review of security models and gives a detailed mathematical description of the property of separability, that is the property that certain software can be treated as though it were being obeyed in isolated computers. The description uses the specification language Z, and the proofs required are given in detail.

END

DATE
FILMED

8 88